CDM

Minimization of Finite State Machines

Klaus Sutner

Carnegie Mellon University

22-minimization 2017/12/15 23:19

Recall our definition of the state complexity $\mathrm{sc}(L)$ of a regular language L: the minimal number of states of any DFA accepting the language.

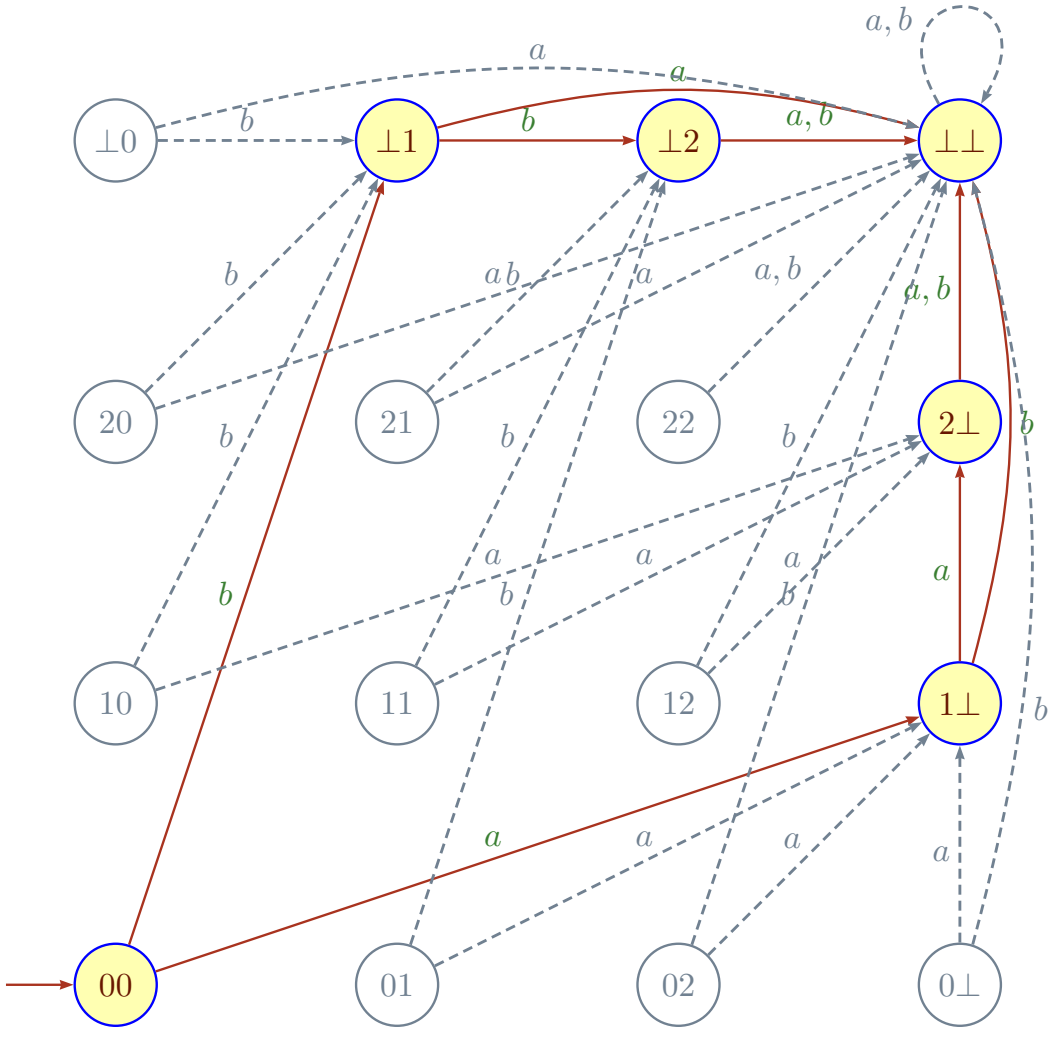
Our next goal is to show how to compute the state complexity of a language: we will construct a corresponding DFA, starting from an arbitrary machine for the language.

As it turns out, the automaton is unique, up to renaming of states. Thus, we have a normal form for any regular language. This is fairly rare, usually there are many canonical descriptions of an object.

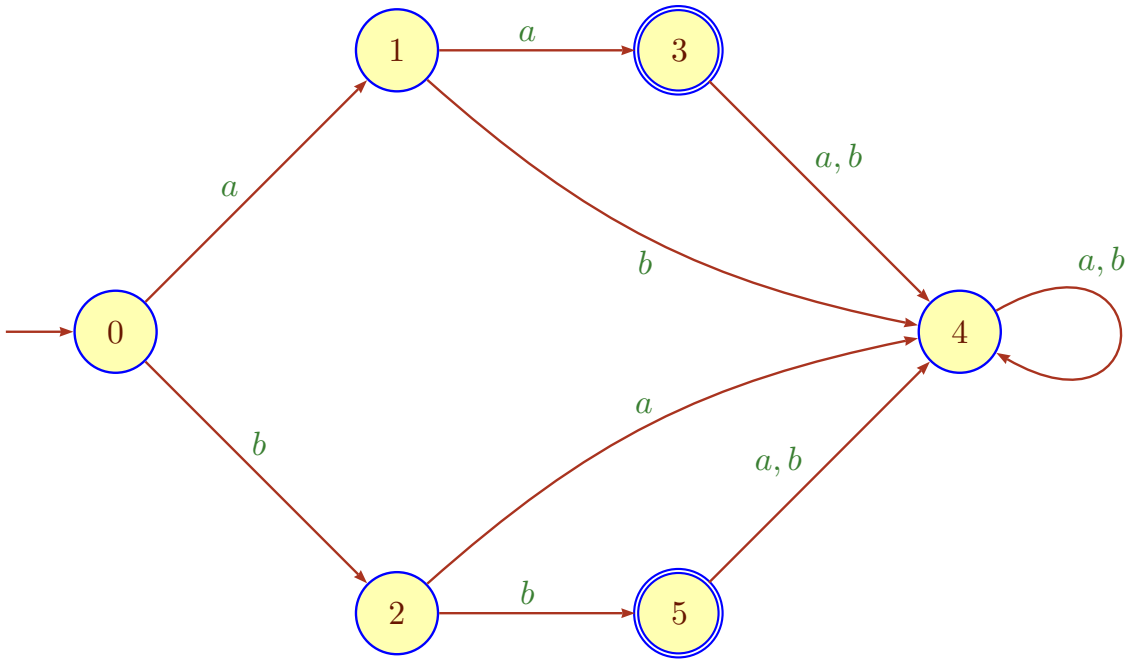
Humans are fairly good at constructing small DFAs that are already minimal–one naturally tends to avoid "useless" states. Unfortunately, this little reassuring fact does not help much:

- Humans fail spectacularly when the machines get large, even a few dozen states are tricky, thousands are not manageable.
- One of the most interesting aspects of finite state machines is that they can be generated and manipulated algorithmically. These algorithm typically do not produce minimal results.

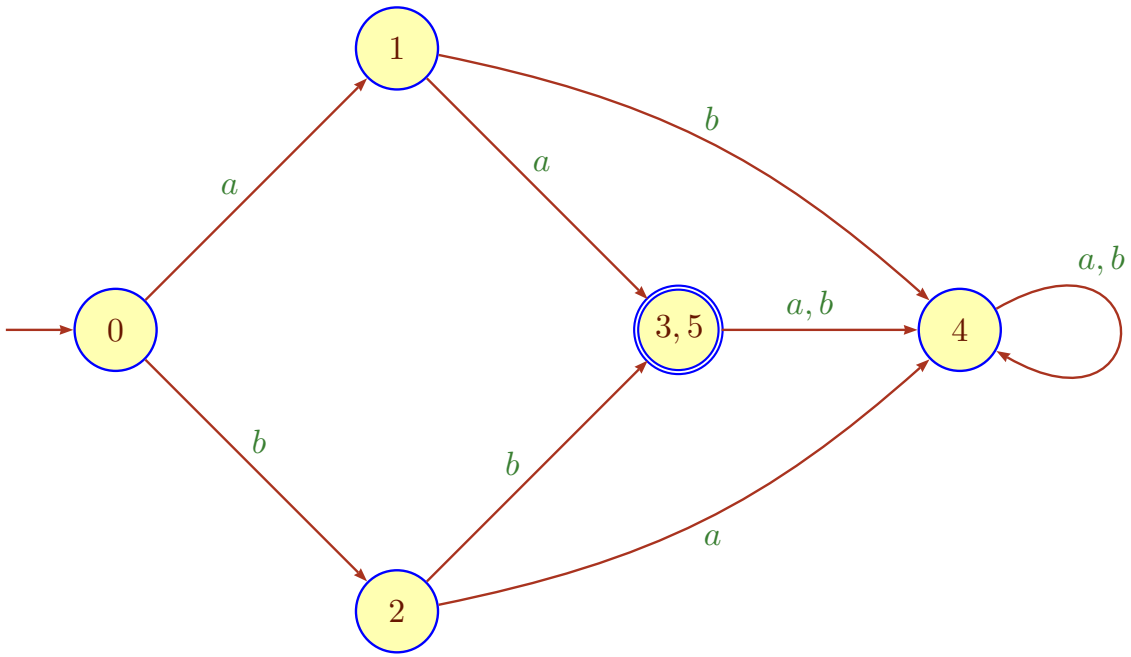
Here is the accessible part of the product automaton for $\{aa, bb\}$, built from the obvious DFAs for aa and bb.



Here is a slightly better diagram for this machine:



However, the state complexity of $\{aa, bb\}$ is only 5 (recall that state complexity is defined in terms of DFAs, so we have to include the sink in the count).



States 3 and 5 are merged into a single state (and the transitions rerouted accordingly).

Definition

A DFA $\mathcal{A}$ is minimal if there is no DFA equivalent to $\mathcal{A}$ with fewer states than $\mathcal{A}$.

Thus the state complexity of $\mathcal{A}$ is the same as the state complexity of $\mathcal{L}(\mathcal{A})$. As already pointed out there are several potential problems with this definition:

- The existence of a minimal DFA is guaranteed by the fact that $\mathbb{N}$ is well-ordered, but there ought to be a more structural reason.
- There might be several minimal DFAs for the same language.
- Even if there is a unique minimal DFA, there might not be a good connection between other DFAs and the minimal one.

How do we know that 5 states are necessary for $\{aa, bb\}$?

- Need initial state q_0.
- Need state $\delta(q_0, a)$ and $\delta(q_0, a) \neq q_0$.
- Need state $\delta(q_0, b)$ and $\delta(q_0, b) \neq q_0, \delta(q_0, a)$.
- Need state $\delta(q_0, aa)$ and $\delta(q_0, aa) \neq q_0, \delta(q_0, a), \delta(q_0, b)$.
- Need state $\delta(q_0, aaa)$ and $\delta(q_0, aa) \neq q_0, \delta(q_0, a), \delta(q_0, b), \delta(q_0, aa)$.

If any of these states were equal the machine would accept the wrong language.

So in a sense all these states are inequivalent, indispensable.

There is no hope to build a machine with fewer than 5 states.

There is an interesting idea hiding in this argument: some states must be distinct, so the machine cannot be too small.

To make this more precise we adopt the following definition.

Definition

Let $\mathcal{A}$ be a DFA. The behavior of a state p is the acceptance language of $\mathcal{A}$ with initial state replaced by p. Two states are (behaviorally) equivalent if they have the same behavior.

In symbols:

$$\begin{aligned}[\![p]\!] &= \mathcal{L}(\langle\, Q, \Sigma, \delta; p, F \,\rangle) \\ &= \{\, x \in \Sigma^\star \mid \delta(p, x) \in F \,\}\end{aligned}$$

So in a DFA the language accepted by the machine is simply $[\![q_0]\!]$.

Whenever $[\![p]\!] = [\![q]\!]$ we can identify p and q: any input that leads to acceptance from p also leads to acceptance from q (and vice versa).

Think about a little daemon sitting in the machine.

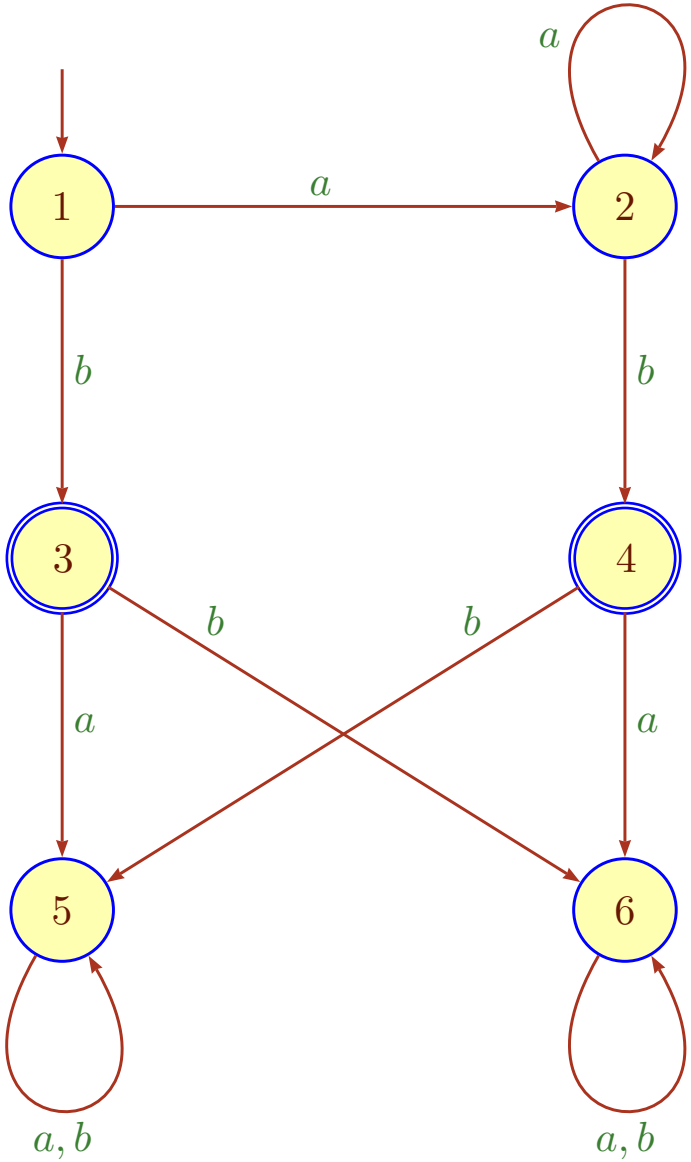
Whenever state p is reached, it may magically flip the state to q, and vice versa. The outside observer would never notice: the daemon infested machine would still accept precisely the same set of strings as the old one.

- So suppose p and p' have the same behavior. We can then collapse p and p' into just one state: to do this we have to redirect all the affected transitions to and from p and q.
- This is easy for the incoming transitions.
- But there is a little problem for the outgoing transitions: one has to merge all equivalent states, not just a few.
- Otherwise the merged states will have nondeterministic transitions emanating from them – and we do not want to deal with nondeterministic machines here.

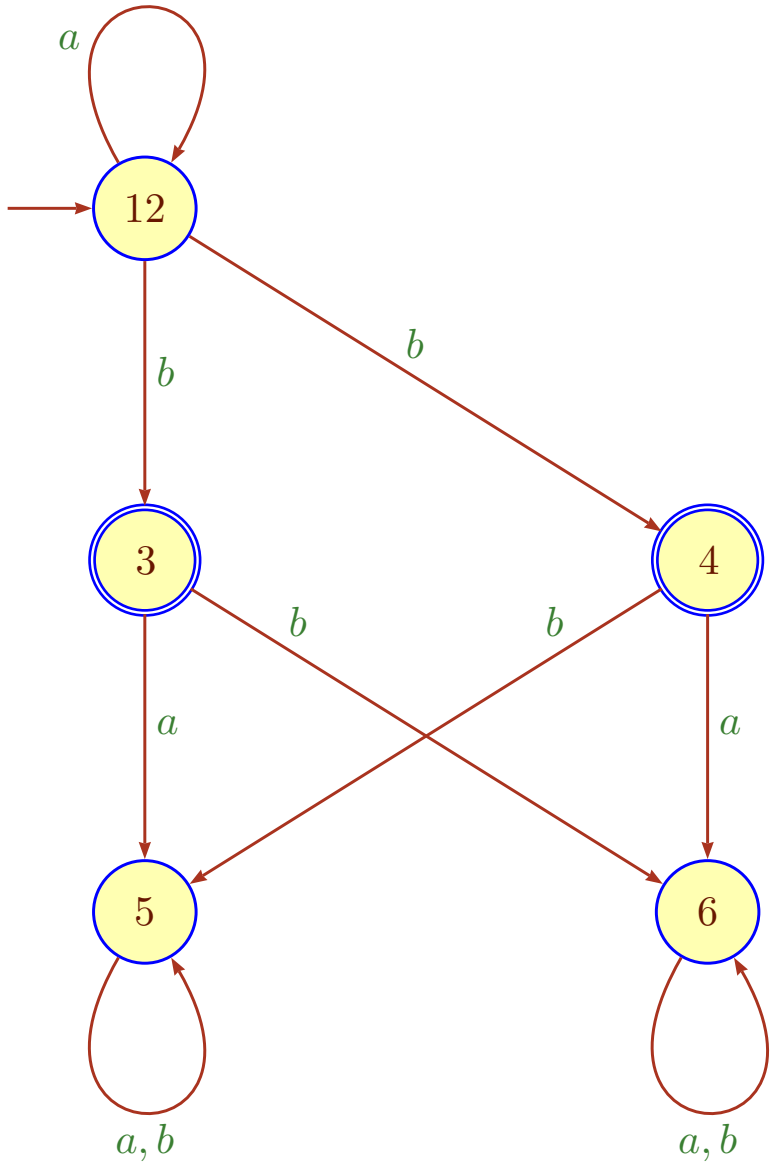
Language:

a^*b.

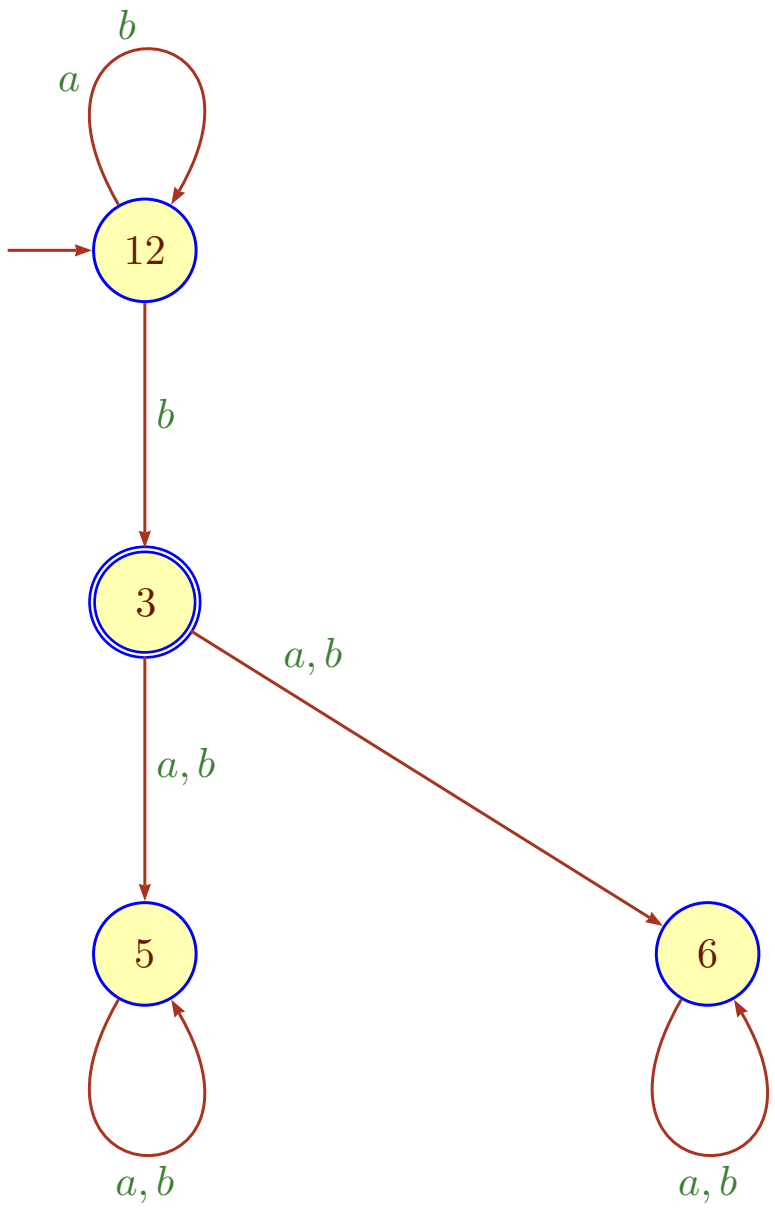
$[\![1]\!] = [\![2]\!]$
$[\![3]\!] = [\![4]\!]$
$[\![5]\!] = [\![6]\!]$



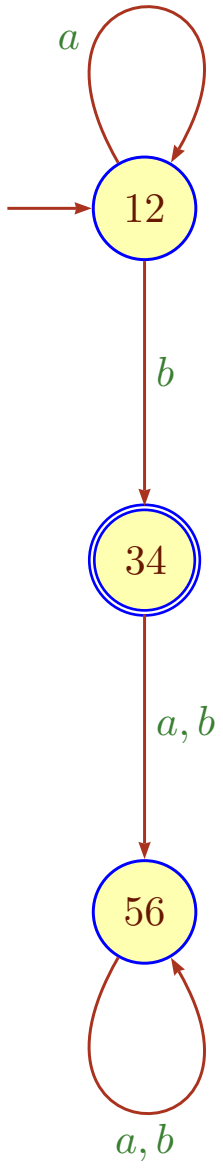
Merging only states 1 and 2 produces a nondeterministic machine.



Merging 1 and 2;
and 3 and 4.



A complete merge produces a DFA.



In the last machine, all states are inequivalent:

$$[\![12]\!] = a^*b$$
$$[\![34]\!] = \varepsilon$$
$$[\![56]\!] = \varnothing$$

So no further state merging is possible.

Definition

A DFA is reduced if all its states are pairwise inequivalent.

Our goal is to establish the following theorem.

Theorem

A DFA is minimal if, and only if, it is accessible and reduced.

Accessibility is computationally cheap. The merging part naturally comes in two phases:

- Determine the required partition of the state set.
- Merge the blocks into single states of the new machine.

The second phase is easy, the first requires work, in particular if one needs fast algorithms.

- Minimal Automata

2. The Algebra of Languages

- The Quotient Machine

- Computing with Equivalences

- Moore's Algorithm

The merging approach is really algebraic in nature. Given some complicated structure $\mathcal{S}$, try to simplify matters as follows:

- Find an equivalence relation E on $\mathcal{S}$,
- that is compatible with the operations on $\mathcal{S}$, and then
- replace $\mathcal{S}$ by the quotient structure $\mathcal{S}/E$.

In general one would like to make the quotient structure as small as possible, so the equivalence relation should be as coarse as possible.

The operations on $\mathcal{S}$ extend naturally to operations on $\mathcal{S}/E$.

The important point here is that not just any equivalence will do, rather we need a congruence: an equivalence that coexists peacefully with the algebraic operations under consideration.

E.g., if $\mathcal{S}$ has a binary operation $*$ then we need

$$a \; E \; b, c \; E \; d \quad \text{implies} \quad a * c \; E \; b * d.$$

Thus, it might be a good idea to take a closer look at the algebra of languages, whatever that may turn out to be.

Example

The classical example is modular arithmetic: the $\mathrm{mod} m$ relation is a congruence with respect to addition and multiplication.

Given an alphabet Σ we consider the carrier set of all languages over Σ:

$$\mathcal{L}(\Sigma) = \mathfrak{P}(\Sigma^\star) = \{\, L \mid L \subseteq \Sigma^\star \,\}$$

Note that $\mathcal{L}(\Sigma)$ is uncountable (same cardinality as the reals) even in the degenerate case $\Sigma = \{a\}$.

From a computational perspective $\mathcal{L}(\Sigma)$ is interesting only as a general framework, we need to restrict our attention to small (countable) subsets of $\mathcal{L}(\Sigma)$ if we want algorithms e.g. for the Membership Problem.

For example, we can study decidable languages in general or easily decidable languages such as regular ones.

Since we are dealing with a powerset, there are the obvious Boolean operations union, intersection and complement that can be applied to languages over Σ. So we have a Boolean algebra

$$\langle \mathcal{L}(\Sigma), \cup, \cap, ^- \rangle$$

That's OK, but not terribly interesting: at no point are we using the fact that the sets in question are sets of words, rather than arbitrary objects.

So the question is: what are interesting operations on $\mathcal{L}(\Sigma)$ that exploit this fact?

Recall that we can "multiply" two words by concatenating them.

We can lift this operations to language by applying concatenation pointwise. Given two languages $L_1, L_2 \subseteq \Sigma^\star$ we concatenate them by concatenating all possible combinations of words:

$$L_1 \cdot L_2 = \{\, xy \mid x \in L_1, y \in L_2 \,\}.$$

Note that this operation fails to commute.

Also, $L \cdot \varnothing = \varnothing \cdot L = \varnothing$ and $L \cdot \{\varepsilon\} = \{\varepsilon\} \cdot L = L$.

Example

$$\{a, b\}\{a, b\} = \{aa, ab, ba, bb\}$$

Boolean operations and concatenation when applied to finite languages produce only finite and co-finite languages and are thus insufficient to generate interesting languages. We need at least one operation that generates an infinite language from a finite one. Here is one such operation that turns out to be immensely useful.

Definition

Let L be a language. The powers of L are the languages obtained by repeated concatenation:

$$L^0 = \{\varepsilon\}$$
$$L^{k+1} = L^k \cdot L$$

The Kleene star of L is the language

$$L^\star = L^0 \cup L^1 \cup L^2 \ldots \cup L^n \cup \ldots$$

Kleene star corresponds roughly to a while-loop or iteration.

Example

$\{a, b\}^\star$: all words over $\{a, b\}$

Example

$\{a, b\}^\star\{a\}\{a, b\}^\star\{a\}\{a, b\}^\star$: all words over $\{a, b\}$ containing at least two a's

Example

$\{\varepsilon, a, aa\}\{b, ba, baa\}^\star$: all words over $\{a, b\}$ not containing a subword aaa

Example

$\{0, 1\}\{0, 1\}^\star$: all numbers in binary, with leading 0's

$\{1\}\{0, 1\}^\star \cup \{0\}$: all numbers in binary, no leading 0's

The choice of concatenation and Kleene star may seem rather arbitrary. It is is justified by the following theorem (see lecture on Kleene algebras for a proof).

Theorem (Kleene)

Every regular language can be obtained from singletons $\{a\}$ *for* $a \in \Sigma$, *and* $\varnothing$, *by finitely many applications of the operations union, concatenation and Kleene star. Given a finite state machine for the language, the decomposition can be generated algorithmically.*

In other words, the collection $\mathrm{Reg}(\Sigma)$ of all regular languages over alphabet Σ is a sub-algebra of

$$\langle\, \mathcal{L}(\Sigma), \cup, \cdot, ^\star, 0, 1 \,\rangle$$

and is generated by singletons $\{a\}$.

Note that the theorem makes a rather surprising claim: it suffices to consider operations union, concatenation and Kleene star when one tries to construct regular languages from atomic pieces (in this case singletons $\{a\}$ and the empty set).

But regular languages are closed under intersection and complement. It is by no means clear how

$$L \cap K \qquad \text{or} \qquad \Sigma^\star - L$$

can be so generated, even if we already know how to handle K and L.

The first part of this result is actually very important in applications: it provides a simple notation system for regular languages.

If we write a for the singleton language $\{a\}$ then all regular language can be written down using just $+$ for union, $\cdot$ for concatenation and * for Kleene star (we won't quibble about the empty set here).

These expressions are usually referred to as regular expressions.

They are crucial for a lot of text searching and manipulation tools such as `grep`, `awk`, `sed` and `pearl`: it is easy to type in regular expressions but would be entirely hopeless to have to input the corresponding finite state machines.

The second part is more of theoretical interest: the algorithm usually generates a really bad decomposition (much too big, but simplification is hard).

Conspicuously absent from our algebra so far is any operation resembling division. If we think of division as the inverse of multiplication (i.e., concatenation) the natural answer is the following.

Definition

Let $L \subseteq \Sigma^\star$ be a regular language and $x \in \Sigma^\star$. The left quotient of L by x is

$$x^{-1} L = \{ y \in \Sigma^\star \mid xy \in L \}.$$

So we are simply removing a prefix x from all words in the language that start with this prefix. If there is no such prefix we get an empty quotient.

This is the reason why it is a bit more elegant to talk about quotients in the context of languages rather than words: for words x and y the quotient $x^{-1}y$ would be undefined whenever x fails to be a prefix of y.

It is standard to write left quotients as

$$x^{-1}\,L$$

Here is the bad news: left quotients are actually a right action of $\Sigma^\star$ on $\mathcal{L}(\Sigma)$.

As a consequence, the first law of left quotients looks backward.

Lemma

Let $a \in \Sigma$, $x, y \in \Sigma^\star$ and $L, K \subseteq \Sigma^\star$. Then the following hold:

- $(xy)^{-1}L = y^{-1}x^{-1}L$,
- $x^{-1}(L \cup K) = x^{-1}L \cup x^{-1}K$,
- $x^{-1}(L \cap K) = x^{-1}L \cap x^{-1}K$,
- $x^{-1}(L - K) = x^{-1}L - x^{-1}K$,
- $a^{-1}(LK) = (a^{-1}L)K \cup \Delta(L)\, a^{-1}K$,
- $a^{-1}L^\star = (a^{-1}L)\, L^\star$.

Here we have used the abbreviation $\Delta(L)$ to simplify notation:

$$\Delta(L) = \begin{cases} \{\varepsilon\} & \text{if } \varepsilon \in L, \\ \varnothing & \text{otherwise.} \end{cases}$$

So $\Delta(L)$ is either zero or one in the language semiring.

Note that $(xy)^{-1}L = y^{-1}x^{-1}L$ and NOT $x^{-1}y^{-1}L$. As already mentioned, the problem is that algebraically left quotients are a right action.

Quotients coexist peacefully with Boolean operations, we can just push the quotients inside.

But for concatenation and Kleene star things are a bit more involved; the lemma makes no claims about the general case where we divide by a word rather than a single letter.

Exercise

Prove the last lemma.

Exercise

Generalize the rules for concatenation and Kleene star to words.

The reason we are interested in quotients is that they are closely related to behaviors of states in a DFA. More precisely, consider the following question:

> What are the possible behaviors of states in an arbitrary DFA for a fixed regular language?

One might think that the behaviors differ from machine to machine, but they turn out to be the same, always.

To see why, first ignore the machines and consider the acceptance language directly. Note that the language is the behavior of the initial state and thus the same in any DFA.

Yet another fixed point: we need to compute the least set $X \subseteq \mathcal{L}(\Sigma)$ such that

- $L \in X$ and
- X is closed under a^{-1} for all $a \in \Sigma$.

The corresponding monotonic operation $F : \mathfrak{P}(\mathcal{L}(\Sigma)) \to \mathfrak{P}(\mathcal{L}(\Sigma))$ (who says types are useless?) is

$$F(X) = X \cup \{\, a^{-1}Y \mid Y \in X, a \in \Sigma \,\}$$

and we are looking for the least fixed point of $\{L\}$ under F. The fixed point exists by Knaster-Tarski.

We write $\mathcal{Q}(L)$ for the set of all quotients of a language L.

Using the lemma, we can compute the quotients of a^*b.

$$a^{-1}\,a^*b = a^*b$$
$$b^{-1}\,a^*b = \varepsilon$$
$$a^{-1}\,\varepsilon = \varnothing$$
$$b^{-1}\,\varepsilon = \varnothing$$
$$a^{-1}\,\varnothing = \varnothing$$
$$b^{-1}\,\varnothing = \varnothing$$

Thus $\mathcal{Q}(a^*b)$ consists of: a^*b, ε and $\varnothing$.

Note that these equations between quotients really determine the transitions in the example machine for state-merging from above.

$$a^{-1}\,a^*b = a^*b \qquad a^*b \xrightarrow{a} a^*b$$

$$b^{-1}\,a^*b = \varepsilon \qquad a^*b \xrightarrow{b} \varepsilon$$

$$a^{-1}\,\varepsilon = \varnothing \qquad \varepsilon \xrightarrow{a} \varnothing$$

$$b^{-1}\,\varepsilon = \varnothing \qquad \varepsilon \xrightarrow{b} \varnothing$$

$$a^{-1}\,\varnothing = \varnothing \qquad \varnothing \xrightarrow{a} \varnothing$$

$$b^{-1}\,\varnothing = \varnothing \qquad \varnothing \xrightarrow{b} \varnothing$$

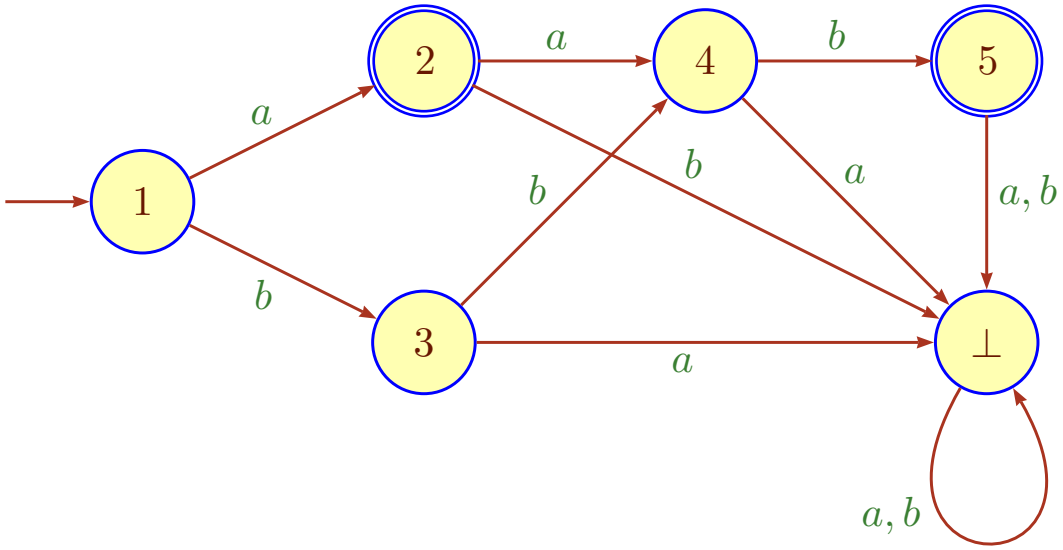
We can also keep track of the words that produce a particular quotient. E.g., let L be the finite language $\{a, aab, bbb\}$.

This time $\mathcal{Q}(L)$ has size 6:

x	$x^{-1} L$
ε	$\{a, aab, bbb\}$
a	$\{\epsilon, ab\}$
b	$\{bb\}$
bb	$\{b\}$
aab	$\{\epsilon\}$
ab	$\varnothing$

Of course the witness x is not uniquely determined, for example $(abz)^{-1} L = (baz)^{-1} L = \varnothing$ for any z. The table lists the length-lex minimal witness in each case.

Moreover, there is a "natural" DFA for L that has six states.



Could this be coincidence? Nah ...

For example, $\delta(1, a) = 2$ and $[\![2]\!] = \{\varepsilon, ab\}$.

Corresponding to $a^{-1} L = \{\varepsilon, ab\}$.

A larger example, $L = L_1 = a^*b^* \cup bab$.

$a^{-1}L_1$	a^*b^*	L_2
$b^{-1}L_1$	$b^* \cup ab$	L_3
$a^{-1}L_2$	L_2	
$b^{-1}L_2$	b^*	L_4
$a^{-1}L_3$	b	L_5
$b^{-1}L_3$	L_4	
$a^{-1}L_4$	$\varnothing$	L_6
$b^{-1}L_4$	L_4	
$a^{-1}L_5$	L_6	
$b^{-1}L_5$	ε	L_7
$a^{-1}L_{6/7}$	L_6	
$b^{-1}L_{6/7}$	L_6	

Exercise

Verify this table.

An even larger example, $L = L_1 = a^*ba^* \cup b^*ab^*$.

$a^{-1}L_1$	$a^*ba^* + b^*$	L_2	$b^{-1}L_5$	b^*	L_8	
$b^{-1}L_1$	$b^*ab^* + a^*$	L_3	$a^{-1}L_6$	b^*		
$a^{-1}L_2$	a^*ba^*	L_4	$b^{-1}L_6$	b^*ab^*		
$b^{-1}L_2$	$a^* + b^*$	L_5	$a^{-1}L_7$	b^*		
$a^{-1}L_3$	$a^* + b^*$		$b^{-1}L_7$	$\varnothing$	L_9	
$b^{-1}L_3$	b^*ab^*	L_6	$a^{-1}L_8$	$\varnothing$		
$a^{-1}L_4$	a^*ba^*		$b^{-1}L_8$	b^*		
$b^{-1}L_4$	a^*	L_7	$a^{-1}L_9$	$\varnothing$		
$a^{-1}L_5$	a^*		$b^{-1}L_9$	$\varnothing$		

Exercise

Verify this table.

Here is a very different example:

$$L = \{ a^i b^i \mid i \geqslant 0 \} = \{\varepsilon, ab, aabb, aaabbb, \ldots\}$$

This time there are infinitely many quotients.

$$\begin{aligned} (a^k)^{-1} L &= \{ a^i b^{i+k} \mid i \geqslant 0 \} \\ (a^k b^l)^{-1} L &= \{b^{k-l}\} && 1 \leqslant l \leqslant k \\ (a^k b^l)^{-1} L &= \varnothing && l > k \end{aligned}$$

This is no coincidence: the language L is not regular.

As always, we can interpret the computation of $\mathcal{Q}(L)$ as a closure operation, albeit one on somewhat complicated types: the ambient set here is $\mathcal{L}(\sigma)$.

$$\mathcal{Q}(L) = \mathsf{Cl}\big(L, (a^{-1} \mid a \in \Sigma)\big)$$

Here we have simply written a^{-1} for the operation $K \mapsto a^{-1}K$.

In a sense, this is just another way of looking at the fixed point operation from above. But it is more appropriate in our context: it is just a somewhat abstract description of an algorithm to compute the quotient.

Wurzelbrunft (who is not much of a hacker) feels very uneasy about this: can we actually implement this, or is it just pure, inherently un-implementable math?

The logical control structure is easy: it's just a while-loop. But we need to represent the basic objects and operations:

- represent languages by some data structure,
- implement the operations $a^{-1}K$,
- implement the equality test $K = K'$.

Are all these problems surmountable?

Naturally, we represent languages by machines (we don't have much choice at this point).

- Since we are only dealing with regular languages we can use DFAs as representation.
- Quotients are then easy to implement: just move the initial state.
- The equality test comes down to checking Equivalence of DFAs, we already know how to do this.

Note how the choice of data structure really settles the whole issue: if a regular language is represented by a DFA we know how to compute quotients, and we know how to check equality.

The question arises: how efficient is this approach?

Suppose the DFA representing L has n states. For simplicity we ignore the size of the alphabet.

Clearly, there will be at most n quotients to compute.

For each one, we have to test equality against $O(n)$ others.

Doing this the obvious way requires $O(n^2)$ steps for each equality check, so each new quotient requires $O(n^3)$ steps.

The whole algorithm is then an unimpressive $O(n^4)$.

Can we speed this up?

Exercise

Explain the running time of this method in detail. Take into account the size of the alphabet.

Before we discuss better algorithms, note that for finite languages

$$L = \{w_1, w_2, \ldots, w_n\}$$

we don't need a DFA to represent the language: we can use any container data structure to represent finite sets of strings. A particularly good choice often is a trie.

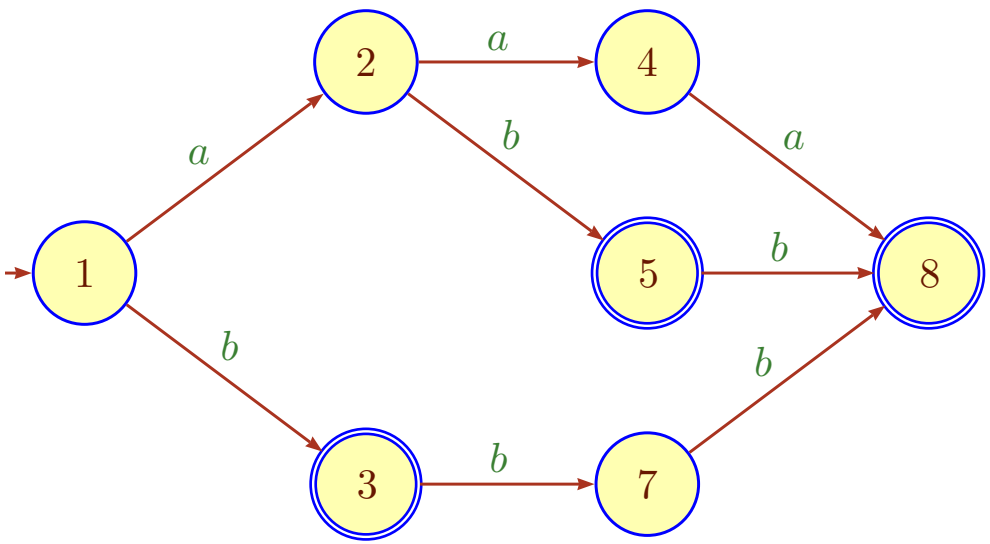
The operations of taking quotients and checking equality are straightforward to implement in any reasonable programming language. Note that except for the sink the diagram of the automaton must be a DAG.

There are better algorithms for maintaining machines for finite languages (hugely important in computational linguistics), but let's not get involved with efficiency at the moment.

As a small example consider $L = \{b, ab, aaa, abb, bbb\}$. Then the 8 quotients are

1	L	5	$\{\varepsilon, b\}$
2	$\{b, aa, bb\}$	6	$\varnothing$
3	$\{\varepsilon, bb\}$	7	$\{b\}$
4	$\{a\}$	8	$\{\varepsilon\}$

corresponding to the partial DFA (the sink 6 is omitted):



- Minimal Automata
- The Algebra of Languages

3. The Quotient Machine

- Computing with Equivalences
- Moore's Algorithm

Here is a simple observation about the relationship between languages (not just regular) and their quotients.

Lemma

Let $L \subseteq \Sigma^\star$ be any language. Then

$$L = \Delta(L) \cup \bigcup_{a \in \Sigma} a \cdot (a^{-1} L)$$

Proof. Duh.

More precisely, a word $x \in L$ is either ε, or it looks like $x = au$ for some $a \in \Sigma, u \in \Sigma^\star$. □

Exercise

Give a fastidious definition of words as functions $w : [n] \to \Sigma$, $n \in \mathbb{N}$, and use this definition to give a formal proof of the Decomposition lemma.

The Decomposition lemma is just about trivial. For $L \subseteq \{a, b\}^\star$ we get

$$L = a \cdot (a^{-1}\, L) \cup b \cdot (b^{-1}\, L) \cup \Delta(L)$$

But, from the right point of view this little observation is quite helpful:

- Think of the quotients as states.
- Then the Decomposition lemma describes the transitions on these states:

$$L \xrightarrow{a} a^{-1} L$$

- The Δ term determines whether a state is final.

In other words, we can build a DFA out of the quotients. To see how, suppose $Q = \mathcal{Q}(L)$ is a finite list of all the quotients of some language L.

Construct a DFA

$$\mathcal{M}_L = \langle\, Q, \Sigma, \delta; q_0, F \,\rangle$$

as follows:

$$\begin{aligned} \delta(K, a) &= a^{-1}\, K \\ q_0 &= L \\ F &= \{\, K \in Q \mid \varepsilon \in K \,\} \end{aligned}$$

This is perfectly in keeping with our definitions: the state set has to be finite, but no one said the states couldn't be complicated.

At any rate, in $\mathcal{M}_L$ we have

$$\delta(q_0, x) = \delta(L, x) = x^{-1}\, L.$$

But then

$$x \in L \iff \varepsilon \in x^{-1}\, L \iff \delta(q_0, x) \in F$$

so that $\mathcal{M}_L$ duly accepts L.

It is clear by now that there is a very close link between behaviors and quotients of the acceptance language.

More precisely, it follows from the Decomposition lemma that in any DFA whatsoever

$$[\![\delta(p,a)]\!] = a^{-1}\,[\![p]\!]$$

Note that it is critical here that DFAs are deterministic: there is only one path in the diagram starting at the initial state labeled by any particular word x.

The theory of nondeterministic machines is much more complicated.

Lemma

Let $\mathcal{A}$ be an arbitrary DFA, p a state and $x \in \Sigma^\star$. Then

$$[\![\delta(p, x)]\!] = x^{-1} [\![p]\!]$$

Proof. Straightforward induction on x. Use

$$(xa)^{-1}L = a^{-1}(x^{-1}L)$$

□

Corollary

Suppose $\mathcal{A}$ is a DFA accepting L. Then for any word x:

$$[\![\delta(q_0, x)]\!] = x^{-1}L$$

Hence all accessible states have as behavior one of the quotients of L. Conversely, all quotients appear as the behavior of at least one state in any DFA for L. This may not sound too impressive, but it has some very interesting consequences.

Corollary

Every regular language has only finitely many left quotients.

Corollary

Every DFA accepting a regular language has at least as many states as the number of quotients of the language.

Corollary

The quotient machine for a regular language has the lowest possible state complexity.

So now we know that for any regular language L the quotient automaton $\mathcal{M}_L$ is minimal:

$$\mathsf{sc}(L) = \#\text{ quotients of } L$$

So, computing state complexity comes down to generating all quotients. We know more or less how to do this algebraically, and we have a clumsy algorithm based on manipulating DFAs.

As we will see later, quotients are often also useful in describing and analyzing finite state machines in general.

Theorem

A DFA for a regular language is minimal with respect to the number of states if, and only if, it is accessible and reduced. Moreover, there is only one such minimal DFA (up to isomorphism): the quotient automaton of the language.

Proof.

Let L be the regular language in question and suppose that L has n quotients.

First assume that $\mathcal{A}$ is an accessible and reduced DFA for L. Then every quotient of L must appear exactly once as the behavior of a state in $\mathcal{A}$, hence $\mathsf{sc}(\mathcal{A}) = n$.

By the corollary every DFA for L has at least n states, so $\mathcal{A}$ is minimal.

For the opposite direction, clearly any minimal automaton $\mathcal{A}$ for L must be accessible.

From the corollary, $\mathsf{sc}(\mathcal{A}) \geqslant n$ and we know how to construct a DFA with exactly n states.

But $\mathcal{A}$ is minimal, so $\mathsf{sc}(\mathcal{A}) = n$.

Again every quotient of L must appear exactly once as the behavior of a state: thus $\mathcal{A}$ is reduced.

It remains to show that all DFAs for L of size n are essentially the same as the quotient machine $\mathcal{M}_L$ – we can rename the states, but other than that the machine is fixed.

To see this note we can define a bijection

$$f : Q \to \mathcal{Q}(L)$$
$$f(p) = [\![p]\!]$$

from the states of $\mathcal{A}$ to the states of $\mathcal{M}_L$ (the quotients of L).

This is a bijection since $\mathcal{A}$ has size n and we know that all quotients must appear as the behavior of at least one state.

Moreover, this bijection is compatible with the transitions in the machines in the sense that $f(\delta(p, a)) = \delta(f(p), a)$. As a diagram:

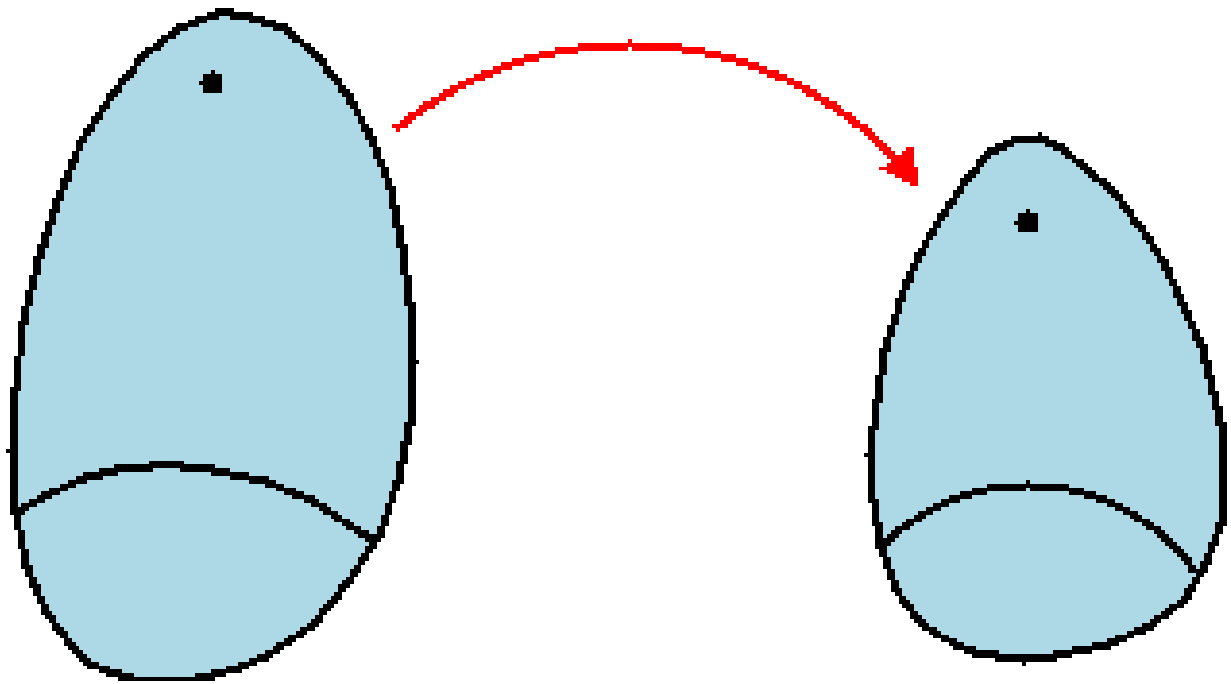
$$\begin{array}{ccc} p & \xrightarrow{a} & \delta(p, a) \\ \downarrow f & & \downarrow f \\ f(p) & \xrightarrow{a} & \delta(f(p), a) \end{array}$$

Lastly, f maps initial to initial, and final to final states.

Hence, the states in $\mathcal{A}$ are just "renamed" quotients: the machines $\mathcal{A}$ and $\mathcal{M}_L$ are isomorphic.

□

The isomorphism from above leads to a more general question: is there a good notion of a structure preserving map between two finite state machines? For simplicity, let's only consider DFAs.



It is clear that for a map f from machine $\mathcal{A}_1$ to machine $\mathcal{A}_2$ to be a homomorphism it must preserve transitions:

$$p \xrightarrow{a} q \qquad \text{implies} \qquad f(p) \xrightarrow{a} f(q)$$

Moreover, we require $f(q_{10}) = q_{20}$ and $f(F_1) = F_2$.

It follows immediately that $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$.

However, we may still have $\mathcal{L}(\mathcal{A}_1) \neq \mathcal{L}(\mathcal{A}_2)$ (why?), so if we are interested in equivalent machines we need to strengthen the conditions a bit:

$$f^{-1}(F_2) = F_1$$

Homomorphisms that have this stronger property and are also surjective are often called covers or covering maps.

Thus, a covering map can identify some states in the first machine while preserving the language.

Needless to say, the classical example of a cover is the behavioral map:

$$f : Q \to \mathcal{Q}(L)$$
$$f(p) = [\![p]\!]$$

Hence we have the following lemma which shows that an arbitrary DFA for a given regular language is always an "inflated" version of the minimal DFA. There always is a close connection between an arbitrary DFA and the minimal automaton.

Lemma

Let L be a regular language and $\mathcal{A}$ an arbitrary accessible DFA for L. Then there is compatible map from $\mathcal{A}$ onto $\mathcal{M}_L$.

There is a natural DFA $\mathcal{A}$ for all words $x \in \{a,b\}^\star$ such that $x_{-3} = a$. The states in $\mathcal{A}$ are words over $\{a,b\}$ of length at most 3 and the transitions are of the form

$$\delta(w,s) = \begin{cases} ws & \text{if } |w| < 3, \\ w_2 w_3 s & \text{otherwise.} \end{cases}$$

The initial state is ε and the final states are $\{aaa, aab, aba, abb\}$. The covering map to the quotient automaton has the form

$$\begin{array}{llll} aaa & \mapsto & aaa & aa, baa \mapsto baa \\ aab & \mapsto & aab & ab, bab \mapsto bab \\ aba & \mapsto & aba & a, ba, bba \mapsto bba \\ abb & \mapsto & abb & \varepsilon, b, bb, bbb \mapsto bbb \end{array}$$

Note that the transition diagram of the minimal automaton is a binary de Bruijn graph (of order 3).

The covering map provides a way to minimize a DFA $\mathcal{A}$: all we need to do is to merge all the states that map to the same quotient: behavioral equivalence is the kernel relation defined by the cover map.

But note that there is a bit of a vicious cycle: to compute the cover f directly we need $\mathcal{M}_L$. If we have the latter there is no need to minimize $\mathcal{A}$.

Nonetheless, covers indicate the right approach to efficient algorithms:

- Start with any DFA $\mathcal{A}$ for L.
- Remove inaccessible states from $\mathcal{A}$.
- Compute the behavioral equivalence relation for $\mathcal{A}$.
- Lastly, merge states with the same behavior.

In principle, all we need to do is to compute the equivalence relation associated with the behavioral map: we already know that, given any accessible DFA $\mathcal{A} = \langle Q, \Sigma; \delta, q_0, F \rangle$, the behavioral equivalence relation E is the kernel relation of the behavior map:

$$[\![.]\!] : Q \to \mathcal{L}(\Sigma), \qquad p \mapsto [\![p]\!]$$

While this mathematically an elegant characterization, it is a bit abstract; we need to determine its computational meaning.

This is very similar to the problem of computing quotients from above: we can represent behaviors as DFAs, we just need to make sure that all the necessary operations can be implemented.

Given a DFA $\mathcal{A} = \langle Q, \Sigma, \delta; q_0, F \rangle$ on n states define

$$\mathcal{A}_p = \langle Q, \Sigma, \delta; p, F \rangle$$

so that $[\![p]\!] = \mathcal{L}(\mathcal{A}_p)$.

We already know how to solve DFA Equivalence in quadratic time, so we can compute behavioral equivalence for $\mathcal{A}$ as follows:

For each pair of states p and q, test if $\mathcal{A}_p$ and $\mathcal{A}_q$ are equivalent.

Of course, the running time looks like $O(n^4)$. That's fine if we are only interested in polynomial time, but it's not enough for an efficient algorithm.

The wary reader will notice that the product machine constructed in DFA Equivalence testing is essentially the same for all p and q: we are really using the square of the transition system underlying $\mathcal{A}$ and we are moving the initial state (p, q) around.

So, we could compute the full product only once and then perform a graph exploration algorithm with different starting points.

Alas, though the coefficients will be better, asymptotically the running time is still $O(n^4)$: each run of, say, DFS is potentially quadratic in n.

Note that $[\![p]\!] = [\![q]\!]$ iff in the full product transition system

$$\mathsf{Cl}((p,q)) \subseteq F_1 \times F_2 \cup (Q_1 - F_1) \times (Q_2 - F_2).$$

Certainly we can compute the points co-reachable from $F_1 \times (Q_2 - F_2) \cup (Q_1 - F_1) \times F_2$ in quadratic time, which suffices to determine behavioral equivalence.

The algorithm is acceptably efficient asymptotically as far as time is concerned, but it is annoying that the space requirement is also quadratic in the size of the original machine.

- Minimal Automata
- The Algebra of Languages
- The Quotient Machine

4. Computing with Equivalences

- Moore's Algorithm

We will switch back and forth between two natural representations of the same concept.

Equivalence Relations

A relation $\rho \subseteq A \times A$ that is reflexive, symmetric and transitive.

Partition

A collection $B_1, B_2, \ldots, B_k$ of pairwise disjoint, non-empty subsets of A such that $\bigcup B_i = A$ (the blocks of the partition).

As always, we need to worry about appropriate data structures and algorithms that operate on these data structures.

Consider an equivalence relation $\rho \subseteq A \times A$.

Suppose $|A| = n$ and $|\rho| = m$, so $n \leqslant m \leqslant n^2$.

data structure	test	space
list of pairs	$O(m)$	$\Theta(m)$
sorted list of pairs	$O(\log n)$	$\Theta(m)$
Boolean matrix	$O(1)$	$\Theta(n^2)$
selector function	$O(1)$	$\Theta(n)$
union/find	$\approx O(1)$	$\Theta(n)$

Only the last two representations are of interest if we are looking for fast algorithms.

Definition

Given a map $f : A \to B$ the kernel relation induced by f is the equivalence relation

$$x \; K_f \; y \iff f(x) = f(y).$$

Note that K_f is indeed an equivalence relation.

This may seem somewhat overly constrained, but in fact every equivalence relation is a kernel relation for some appropriate function f: just let $f(x) = [x]$. The codomain here is $\mathfrak{P}(A)$ which is not attractive computationally.

But, we can use a function $f : A \to A$: just choose a fixed representative in each class $[x]$.

In general we need to assume the existence of such a choice function axiomatically, but in any context relevant to us things are much simpler: we can always assume that A carries some natural total order.

In fact, usually $A = [n]$ and we can store f as a simple array: this requires only $O(n)$ space and equivalence testing is $O(1)$ with very small constants.

Definition

The canonical selector function for an equivalence relation R on A is

$$\mathsf{sel}_R(x) = \min\big(\, z \in A \mid x \;\rho\; z \,\big)$$

So each equivalence class is represented by its least element.

To test whether $a, b \in A$ are equivalent we only have to compute $f(a)$ and $f(b)$ and test for equality. If the values of f are stored in an array this is $O(1)$, with very small constants.

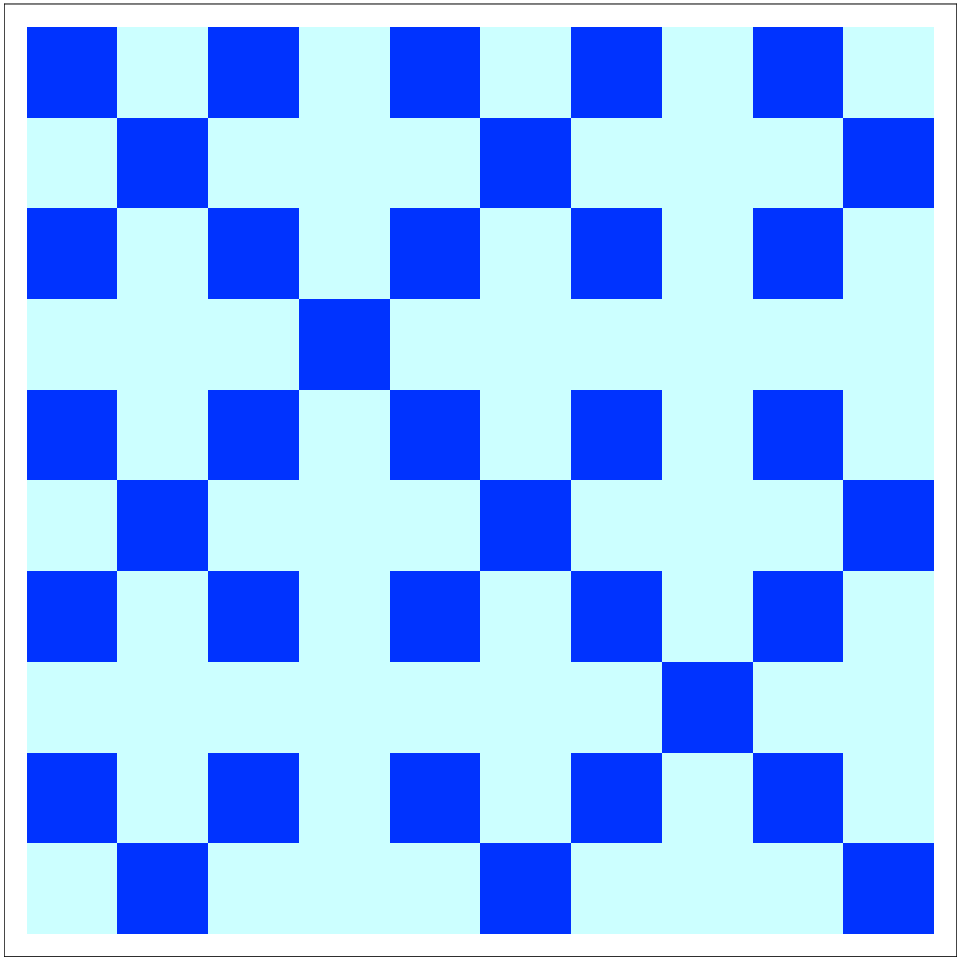
Consider the equivalence relation E on $[10]$ with blocks

$$(1,3,5,7,9),(2,6,10),(4),(8)$$

List of pairs

$$(1,1),(1,3),(1,9),\ldots,(9,1),(2,2),(2,6),\ldots,(10,2),(4,4),(8,8)$$

Sorted list of pairs: well, just sort the thing ...



A pretty picture of a Boolean matrix.

Kernel representation

Note that $p\ E\ q \iff \nu_2(p) = \nu_2(q)$ where $\nu_2(x) = \max\big(\ k \mid 2^k$ divides $x\ \big)$.

Hence E is the kernel relation of ν_2.

The canonical selector function is

$$(1, 2, 1, 4, 1, 2, 1, 8, 1, 2)$$

Definition

Let E be an equivalence relation on A and $B \subseteq A$. Then E saturates B if B is the union of equivalence classes of E.

In other words,

$$B = \bigcup_{x \in B} [x]_E.$$

Proposition

In any DFA, the behavioral equivalence relation saturates the set of final states.

This means that behavioral equivalence is a refinement of the basic partition $(F, Q - F)$.

We can use this as the starting point in an approximation algorithm.

Here are some basic ideas involving equivalence relations.

Definition

Let ρ and σ be two equivalence relations on A. ρ is finer than σ (or: σ is coarser that ρ), in symbols $\rho \sqsubseteq \sigma$, if $x \; \rho \; y$ implies $x \; \sigma \; y$.

To avoid linguistic dislocations, we mean this to include the case where ρ and σ are the same. We will say that ρ is strictly finer than σ if we wish to exclude equality.

In terms of blocks this means that every block of ρ is included in a block of σ.

If we think of equivalence relations as sets of pairs then

$$\rho \sqsubseteq \sigma \iff \rho \subseteq \sigma.$$

We also need some simple manipulations of equivalence relations.

Definition (Meet of Equivalence Relations)

Let ρ and σ be two equivalence relations on A. Then $\rho \sqcap \sigma$ denotes the coarsest equivalence relation finer than both ρ and σ.

In other words,

$$x \, (\rho \sqcap \sigma) \, y \iff x \, \rho \, y \wedge x \, \sigma \, y.$$

This is sometimes written $\rho \cap \sigma$ which is fine if we think of the relations as sets of pairs, but a bit misleading otherwise.

The dual notion of meet is join.

Definition (Join of Equivalence Relations)

Let ρ and σ be two equivalence relations on A. Then $\rho \sqcup \sigma$ denotes the finest equivalence relation coarser than both ρ and σ.

Note that $\rho \sqcup \sigma$ is required to be an equivalence relation, so we cannot in general expect $\rho \sqcap \sigma = \rho \cup \sigma$ in the sets-of-pairs model: the union typically fails to be transitive. Hence, we have to take the transitive closure:

$$\rho \sqcup \sigma = \mathsf{tcl}(\rho \cup \sigma)$$

Let's take a closer look at the problem of computing the meet of two equivalence relations.

We may safely assume that the carrier set is $A = [n]$ and that both relations ρ and σ are given by their canonical selectors (implemented as two arrays r and s of size n).

Let $\tau = \rho \sqcap \sigma$. Then

$$p \; \tau \; q \iff \mathsf{sel}_\rho(p) = \mathsf{sel}_\rho(q) \wedge \mathsf{sel}_\sigma(p) = \mathsf{sel}_\sigma(q)$$

so we are really looking for identical pairs in the table

1	2	3	$\ldots$	p	$\ldots$	n
$r(1)$	$r(2)$	$r(3)$	$\ldots$	$r(p)$	$\ldots$	$r(n)$
$s(1)$	$s(2)$	$s(3)$	$\ldots$	$s(p)$	$\ldots$	$s(n)$

The selector t for τ then looks like so:

	1	2	3	4	5	6	7	8
r	1	1	1	1	5	5	1	5
s	1	2	2	2	1	1	1	2
t	1	2	2	2	5	5	1	8

```
//  construct meet R and R^a
  for( p = 1 .. n ) {
       i = r[p];                    // selector for R
       j = r[delta[p,a]];           // selector for R_a
       if( (i,j) is new  )
            t[p] = val(i,j) = p;
       else
            t[p] = val(i,j);
  }
```

The algorithm uses only trivial data structures except for the "new" query: we have to check if a pair has already been encountered.

The natural choice here is a hash table, though other fast container types are also plausible.

Proposition

Using array representations, we can compute the meet of two equivalence relations in expected linear time.

- Minimal Automata
- The Algebra of Languages
- The Quotient Machine
- Computing with Equivalences

5. Moore's Algorithm

This method goes back to a paper by E. F. Moore from 1956.

The main idea is to start with the very rough approximation $(F, Q - F)$ and then refine this equivalence relation till we get behavioral equivalence.

More precisely, consider all the maps $\mathcal{F} = \{ \delta_a \mid a \in \Sigma \}$.

We need the coarsest equivalence relation finer than $(F, Q - F)$ that is compatible with respect to $\mathcal{F}$.

The constraint "coarsest" is important, otherwise we could just refine ρ to the identity.

Definition

Let $f : A \to A$ be an endofunction and $\mathcal{F}$ a family of such functions.

An equivalence relation ρ on A is f-compatible if $x \; \rho \; y$ implies $f(x) \; \rho \; f(y)$.

ρ is $\mathcal{F}$-compatible if it is f-compatible for all $f \in \mathcal{F}$.

Let ρ be some equivalence relation and write $\rho^{\mathcal{F}}$ for the coarsest refinement of ρ that is $\mathcal{F}$-compatible. Note that

$$\rho^{\mathcal{F}} = \bigsqcup \{ \, \sigma \sqsubseteq \rho \mid \sigma \; \mathcal{F}\text{-compatible} \, \}$$

Of course, we need a real algorithm to compute this join.

To compute $\rho^{\mathcal{F}}$ first define for any $f \in \mathcal{F}$ and any equivalence relation σ:

$$p \; \sigma_f \; q \Leftrightarrow f(p) \; \sigma \; f(p)$$

$$R_f(\sigma) = \sigma \sqcap \sigma_f$$

It is easy to see that $R_f(\sigma)$ is indeed an equivalence relation and is a refinement of σ. The following lemma shows that we cannot make a mistake by applying these refinement operations.

Lemma

- $\rho^{\mathcal{F}} \sqsubseteq \sigma \sqsubseteq \rho$ *implies Let* $\rho^{\mathcal{F}} \sqsubseteq R_f(\sigma) \sqsubseteq \sigma$ *for all* $f \in \mathcal{F}$.
- $\rho^{\mathcal{F}} \sqsubseteq \sigma \sqsubseteq \rho$, σ *not* $\mathcal{F}$*-compatible implies* $R_f(\sigma) \sqsubsetneq \sigma$ *for some* $f \in \mathcal{F}$.

Let $\tau \sqsubseteq \rho$ be $\mathcal{F}$-compatible and assume $x\ \tau\ y$. By assumption, $\tau \sqsubseteq \sigma$. By compatibility, $f(x)\ \tau\ f(y)$, whence $f(x)\ \sigma\ f(y)$. But then $x\ R_f(\sigma)\ y$.

Since σ fails to be $\mathcal{F}$-compatible there must be some $f \in \mathcal{F}$ such that $x\ \sigma\ y$ but not $f(x)\ \sigma\ f(y)$. Hence $R_f(\sigma) \neq \sigma$.

□

According to the lemma, we can just apply the operations R_f repeatedly until we get down to $\rho^{\mathcal{F}}$.

Surprise, surprise, this is Yet Another Fixed Point problem. Let

$$R(\rho) = \prod_{f \in \mathcal{F}} R_f(\rho)$$

Then behavioral equivalence is the fixed point of $(F, Q - F)$ under R.

Alas, this giant-step method is not a good approach algorithmically, it is preferable to perform a sequence of k baby-steps $\rho \mapsto R_f(\rho)$ and cycle through the functions $f = \delta_a$.

Exercise

Show how to streamline the fixed point algorithm by running through a sequence of baby refinement steps.

Once we have computed the behavioral equivalence relation E (or, for that matter, any other compatible equivalence relation on Q) we can determine the quotient structure: we replace Q by Q/E, and q_0 and F by the corresponding equivalence classes.

Define

$$\delta'([p]_E, a) = [\delta(p, a)]_E$$

Proposition

This produces a new DFA that is equivalent to the old one, and reduced.

Exercise

Show that this merging really produces a DFA (rather than some random finite state machine.

As we have seen, each refinement step is $O(n)$, so a big step is $O(kn)$ where k is the cardinality of the alphabet.

Thus the running time will be $O(knr)$ where r is the number of refinement rounds. In many cases r is quite small, but one can force $r = n - 2$.

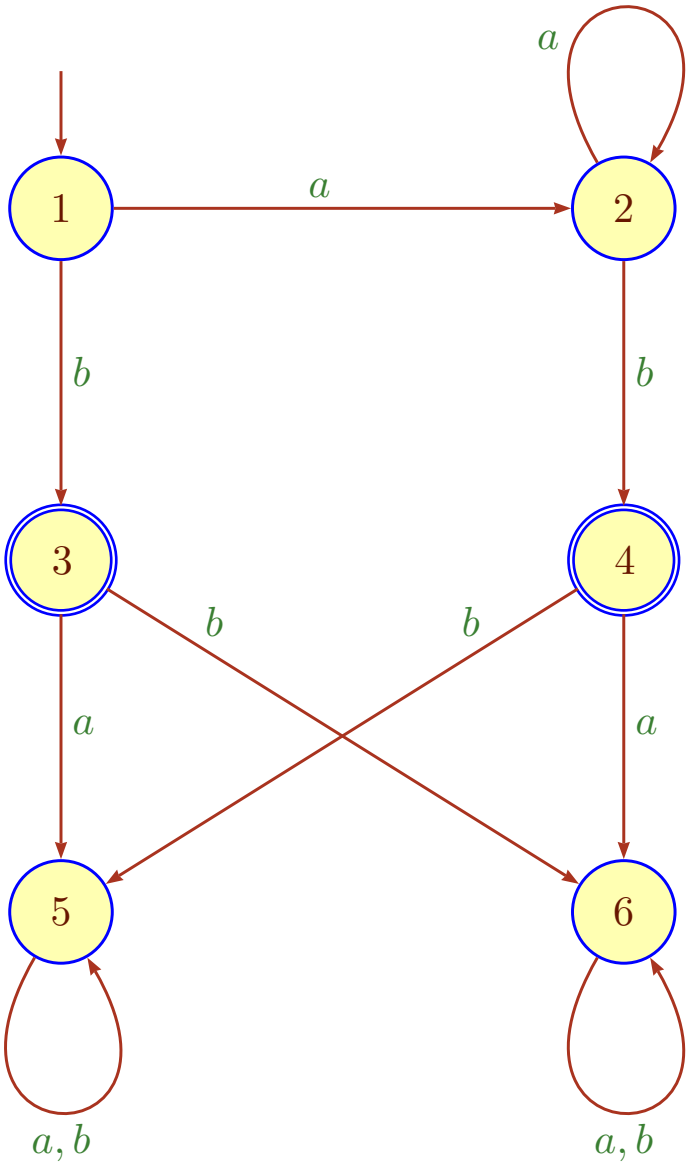
Lemma

Moore's minimization algorithm runs in (expected) time $O(kn^2)$.

Exercise

Figure out how to guarantee linear time for each stage at the cost of a quadratic time initialization. Discuss advantages and disadvantages of this method.

The 6-state DFA for $a^\star b$.



Transition matrix

	1	2	3	4	5	6
a	2	2	5	6	5	6
b	3	4	6	5	5	6

final states $\{3, 4\}$:

	1	2	3	4	5	6
E_0	1	1	3	3	1	1
a	1	1	1	1	1	1
b	3	3	1	1	1	1
E_1	1	1	3	3	5	5
a	1	1	5	5	5	5
b	3	3	5	5	5	5
E_2	1	1	3	3	5	5

Hence $E_2 = E_1$ and the algorithm terminates. Merged states are $\{1, 2\}$, $\{3, 4\}$, $\{5, 6\}$.

To save space, we have performed giant refinement steps.

Consider the DFA with final states $\{1, 4\}$ and transition table

	1	2	3	4	5	6	7	8
a	2	4	5	2	6	8	4	6
b	3	5	4	3	7	4	8	7

produces the trace:

	1	2	3	4	5	6	7	8
E_0	1	2	2	1	2	2	2	2
a	2	1	2	2	2	2	1	2
b	2	2	1	2	2	1	2	2
E_1	1	2	3	1	5	3	2	5
a	2	1	5	2	3	5	1	3
b	3	5	1	3	2	1	5	2
E_2	1	2	3	1	5	3	2	5

The last minimization method may be the most canonical, but there are others. Noteworthy is in particular a method by Brzozowski that uses reversal and Rabin-Scott determinization to construct the minimal automaton.

Write

- $\mathrm{rev}(\mathcal{A})$ for the reversal of any finite state machine, and
- $\mathrm{pow}(\mathcal{A})$ for the accessible part obtained by determinization.

Thus pow preserves the acceptance language but rev reverses it.

Lemma

If $\mathcal{A}$ is an accessible DFA, then $\mathcal{A}' = \mathsf{pow}(\mathsf{rev}(\mathcal{A}))$ is reduced.

Proof.

Let $\mathcal{A} = \langle\, Q, \Sigma, \delta; q_0, F \,\rangle$.

$\mathcal{A}'$ is accessible by construction, so we only need to show that any two states have different behavior.

Let $P = \delta_x^{-1}(F) \neq P' = \delta_y^{-1}(F)$ in $\mathcal{A}'$ for some $x, y \in \Sigma^\star$.

We may safely assume that $p \in P - P'$.

Since $\mathcal{A}$ is accessible, there is a word z such that $p = \delta_z(q_0)$.

Since $\mathcal{A}$ is deterministic, z^{op} is in the $\mathcal{A}'$-behavior of P but not of P'.

□

On occasion the last lemma can be used to determine minimal automata directly.

For example, if $\mathcal{A} = \mathcal{A}_{a,-k}$ is the canonical NFA for the language "kth symbol from the end is a", then rev(pow(rev($\mathcal{A}$))) is $\mathcal{A}$ plus a sink. Hence pow($\mathcal{A}$) must be the minimal automaton.

The same holds for the natural DFA $\mathcal{A}$ that accepts all words over $\{0, 1\}$ whose numerical values are congruent 0 modulo some prime p. Then rev($\mathcal{A}$) is again an accessible DFA and pow(rev(pow(rev($\mathcal{A}$)))) is isomorphic to $\mathcal{A}$.

More generally, we can use the lemma to establish the following surprising minimization algorithm.

Theorem (Brzozowski 1963)

Let $\mathcal{A}$ be a finite state machine. Then the automaton pow(rev(pow(rev($\mathcal{A}$)))) *is (isomorphic to) the minimal automaton of $\mathcal{A}$.*

Proof.

$\widehat{\mathcal{A}} =$ pow(rev($\mathcal{A}$)) is an accessible DFA accepting $\mathcal{L}(\mathcal{A})^{\mathrm{op}}$.

By the lemma, $\mathcal{A}' =$ pow(rev($\widehat{\mathcal{A}}$)) is the minimal automaton accepting $\mathcal{L}(\mathcal{A})^{\mathrm{op}\,\mathrm{op}} = \mathcal{L}(\mathcal{A})$.

□

One might ask whether Moore or Brzozowski is better in the real world. Somewhat surprisingly, given a good implementation of Rabin-Scott determinization, there are some examples where Brzozowski's method is faster.

Theorem (David 2012)

Moore's algorithm has expected running time $O(n \log \log n)$.

Theorem (Felice, Nicaud, 2013)

Brzozowski's algorithm has exponential expected running time.

These results assume a uniform distribution, it is not clear whether this properly represents "typical" inputs.